RPi.GPIO basics 1 – how to check what RPi.GPIO version you have

raspi.tv/2013/rpi-gpio-basics-1-how-to-check-what-rpi-gpio-version-you-have

It struck me the other day that I've published some fairly advanced RPi.GPIO tutorials, (e.g. interrupts and PWM) but not done anything more basic, apart from the Gertboard examples. So here's an attempt to remedy that situation.

Over the next few days, we're going to have a walk around Ben Croston's RPi.GPIO, which is now at version 0.5.3a. Some of this stuff may be new to you even if you've been using RPi.GPIO's more advanced features, so it's worth having a look.

What is RPi.GPIO?

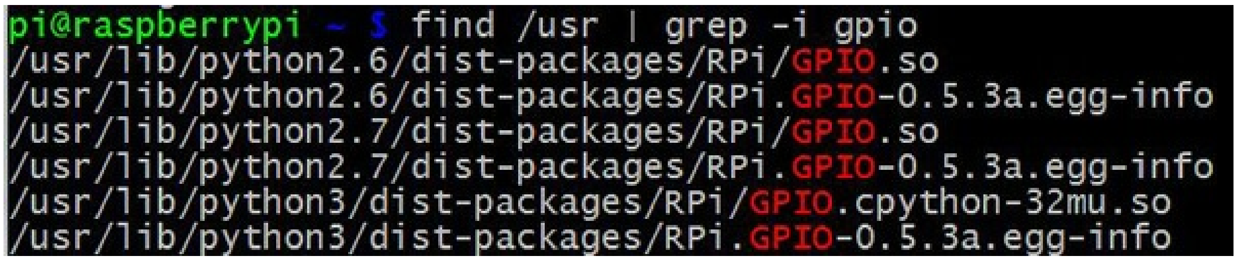
It's an easy way of controlling the Pi's **G**eneral **P**urpose **I**nput **O**utput ports in Python. Control the world with your Raspberry Pi.

How to check what RPi.GPIO version you have

This works for all versions of RPi.GPIO

```
find /usr | grep -i
gpio
```

And the output will look something like this (although there will be more of it)…



```
pi@raspberrypi ~ $ find /usr | grep -i gpio
/usr/lib/python2.6/dist-packages/RPi/GPIO.so
/usr/lib/python2.6/dist-packages/RPi.GPIO-0.5.3a.egg-info
/usr/lib/python2.7/dist-packages/RPi/GPIO.so
/usr/lib/python2.7/dist-packages/RPi.GPIO-0.5.3a.egg-info
/usr/lib/python3/dist-packages/RPi/GPIO.cpython-32mu.so
/usr/lib/python3/dist-packages/RPi.GPIO-0.5.3a.egg-info
```

You can see all those lines with **0.5.3a.egg-info** telling me I have version 0.5.3a.

How to check RPi.GPIO version in Python (works with 0.4.1a +)

While the above works without even using Python, from RPi.GPIO 0.4.1 onwards (September 2012), you can check the version in your Python programs.

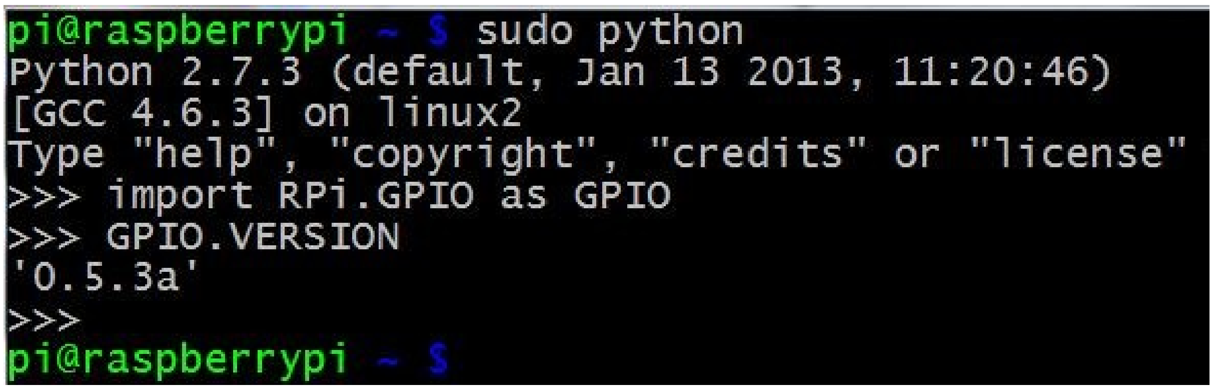
It's quite useful to be able to find out what version of RPi.GPIO you have. Some features require a recent version, whereas others work with older versions. For example, interrupts were added at 0.5.1, PWM at 0.5.2 and with 0.5.3 some bugs have been squashed.

You don't have to write or run any programs to find out which version you have. You can do it very quickly in a live Python session. If you don't know how to do that, it's a great way of testing things out in Python.

Running a live Python session

In the command line environment (either before typing startx or, if you're in LXDE, start LXTerminal) type…

`sudo python` run python as super user (needed for RPi.GPIO to work)
Ignore the "help" "copyright" "credits" "license" notice. Just type the next line at the >>> prompt
`import RPi.GPIO as GPIO` loads the RPi.GPIO library so we can use it
`GPIO.VERSION` this returns the version number of RPi.GPIO
`CTRL+D` ends the python session cleanly (it won't appear on the screen).



```
pi@raspberrypi ~ $ sudo python
Python 2.7.3 (default, Jan 13 2013, 11:20:46)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license"
>>> import RPi.GPIO as GPIO
>>> GPIO.VERSION
'0.5.3a'
>>>
pi@raspberrypi ~ $
```

Watch out for RPi.GPIO. **The i in RPi, is lowercase.** All the rest is uppercase. Get it wrong and it won't work.

So next time someone publishes a python script "that requires RPi.GPIO 0.5.3 or higher", you'll know how to check which version you have.

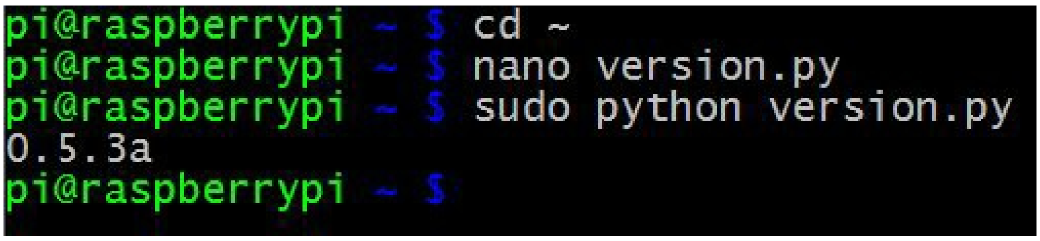
If you want to make it into a little program…

```
import RPi.GPIO as GPIO
a = GPIO.VERSION
print a
```

If you don't like live python environments, you could make it into a little script using the code above.
`cd ~`
`nano version.py`
copy and paste or retype the above script
`CTRL+O` save
`ENTER` confirm
`CTRL+X` exit

Then, to run it, type `sudo python version.py`

…and this is what you should see…



```
pi@raspberrypi ~ $ cd ~
pi@raspberrypi ~ $ nano version.py
pi@raspberrypi ~ $ sudo python version.py
0.5.3a
pi@raspberrypi ~ $
```

How to update your RPi.GPIO

If you find your RPi.GPIO version is horribly out of date, you can update it to the latest version with…

```
sudo apt-get update && sudo apt-get
upgrade
```

But, if you haven’t updated for a while it could take quite a long time to update all your packages (>1hr).

If you don’t want to do it that way, you could use this, which updates your package list and installs the latest RPi.GPIO

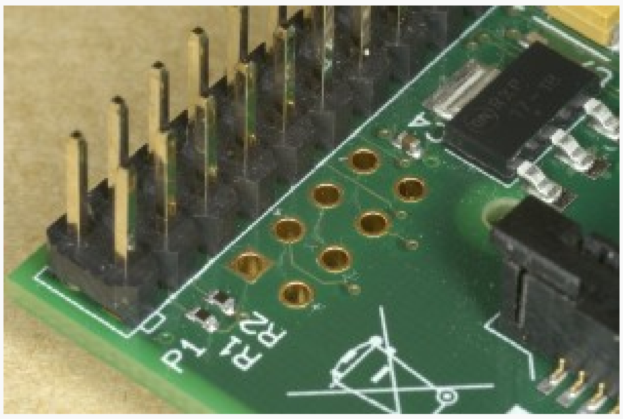
```
sudo apt-get update && sudo apt-get install python-rpi.gpio python3-
rpi.gpio
```

RPi.GPIO basics 2 – how to check what Pi board Revision you have

raspi.tv/2013/rpi-gpio-basics-2-how-to-check-what-pi-board-revision-you-have

It all started in September 2012. The Raspberry Pi Foundation went and made some improvements to the Pi. How dare they!! ;) Seriously, though, there were some significant improvements to the Pi and a Rev 2 version was launched with double the RAM and some other changes/additions/improvements.

This threw up some minor headaches for developers because some of the GPIO pinouts were changed and some new GPIO ports were made available on a brand new "solder it yourself if you want it" header called P5 (see the leaning header of Pi5a)


8 holes for P5 header on Raspberry Pi Rev 2. Square = pin 1

GPIO 0 became GPIO 2 (P1 pin 3)
GPIO 1 became GPIO 3 (P1 pin 5)
GPIO 21 became GPIO 27 (P1 pin 13)

The two i^{2}c buses were swapped around:
SDA0 became SDA1 (P1 pin 3 alternative function)
SCL0 became SCL1 (P1 pin 5 alternative function)

Some new GPIO ports 28, 29, 30, 31 were made available if you solder on P5 header.

(Update Dec 2014: We also have the Compute Module, the B+ and the A+ now, so it's really useful to be able to check what sort of Pi a program is running on.)

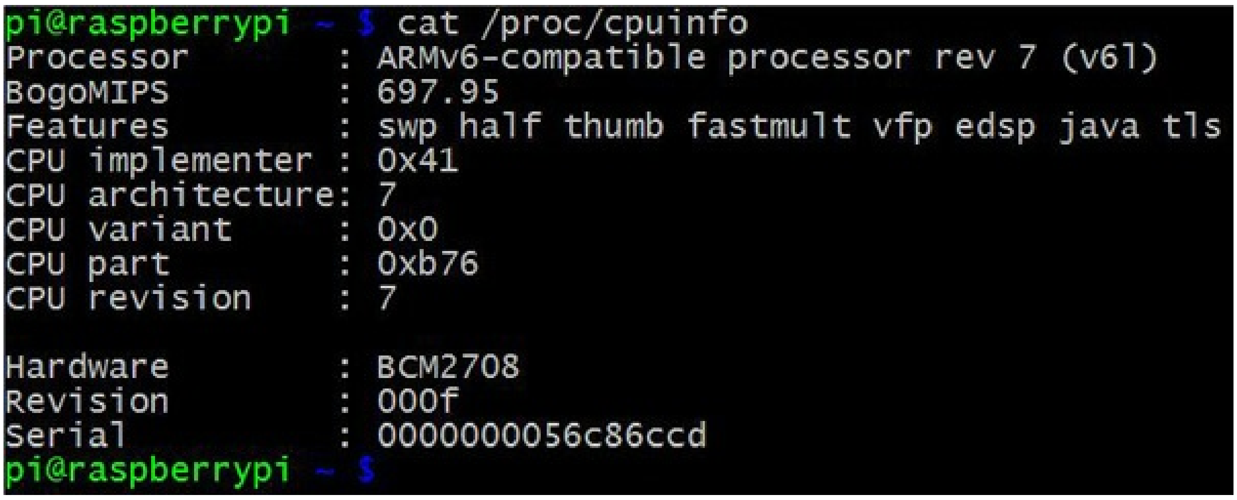
Who cares?

People whose programs no longer work with all Raspberry Pis – that's who!

For example, the Gertboard LEDs program uses GPIO21. If using a Rev 1 Pi this worked fine, but on a Rev 2 Pi (including model A) the fifth LED wouldn't light because GPIO21 is now GPIO27. To work properly, the program needs to be able to know whether it's running on a Rev 1 or Rev 2 Pi, so it can use the correct ports.

How to check your Raspberry Pi Revision number?

There's a way to see what your Pi Revision is…
```
cat
/proc/cpuinfo
```



```
pi@raspberrypi ~ $ cat /proc/cpuinfo
Processor       : ARMv6-compatible processor rev 7 (v6l)
BogoMIPS        : 697.95
Features        : swp half thumb fastmult vfp edsp java tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xb76
CPU revision    : 7

Hardware        : BCM2708
Revision        : 000f
Serial          : 0000000056c86ccd
pi@raspberrypi ~ $
```

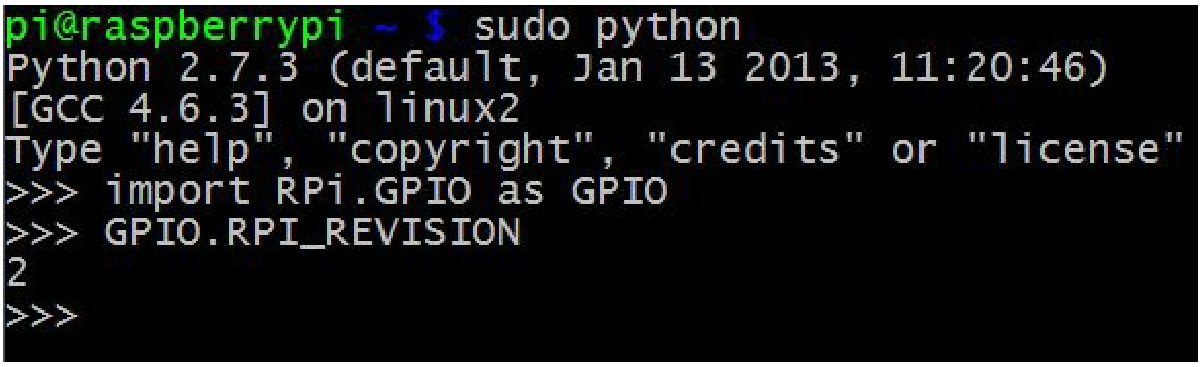
You can see this gives lots of info. Near the bottom is " **Revision : 000f**"

This tells me I have a Rev 2 Pi, but it's a bit clunky. There are several different rev. codes for different Pi models and manufacturers (here's a list of them). We could write some code to check the cpuinfo and extract the bit we want, compare it with known revision codes etc. But we don't need any of that because, from RPi.GPIO 0.4.0a onwards (September 2012) we can use a built-in RPi.GPIO variable which does it all for us. (If you have older than 0.4.0a, see yesterday's post for updating instructions.)

It's called **GPIO.RPI_REVISION**

As before, we can try this out in a live python session…

```
sudo
python
import RPi.GPIO as
GPIO
GPIO.RPI_REVISION
```



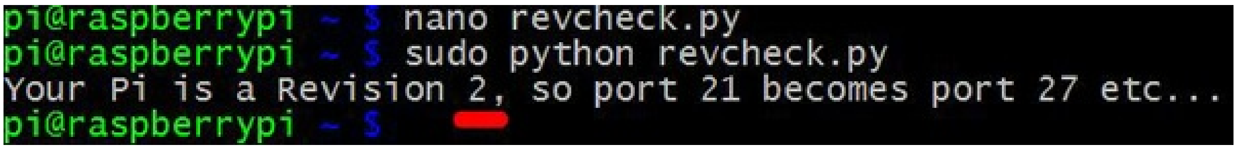
```
pi@raspberrypi ~ $ sudo python
Python 2.7.3 (default, Jan 13 2013, 11:20:46)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license"
>>> import RPi.GPIO as GPIO
>>> GPIO.RPI_REVISION
2
>>>
```

Mine returns a 2 because it's a Rev 2 Pi.

Possible answers are 0 = Compute Module, 1 = Rev 1, 2 = Rev 2, 3 = Model B+/A+

Or you can incorporate the code into a program as we did in yesterday's post …

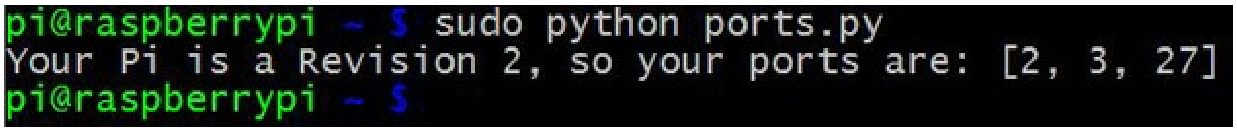
```
import RPi.GPIO as GPIO
print "Your Pi is a Revision %s, so port 21 becomes port 27 etc..." %
GPIO.RPI_REVISION
```



```
pi@raspberrypi ~ $ nano revcheck.py
pi@raspberrypi ~ $ sudo python revcheck.py
Your Pi is a Revision 2, so port 21 becomes port 27 etc...
pi@raspberrypi ~ $
```

So then, when you're doing some GPIO work that uses any of the ports 0, 1, 2, 3, 21, 27, 28, 29, 30, 31, you can be sure to control the correct ports. See the example below…

```
import RPi.GPIO as GPIO
if GPIO.RPI_REVISION == 1:
    ports = [0, 1, 21]
else:
    ports = [2, 3, 27]
print "Your Pi is a Revision %s, so your ports are: %s" % (GPIO.RPI_REVISION,
ports)
```



```
pi@raspberrypi ~ $ sudo python ports.py
Your Pi is a Revision 2, so your ports are: [2, 3, 27]
pi@raspberrypi ~ $
```

As long as no further changes to GPIO pin allocations are made, the above should work with all future Pi revisions. If further changes are made, they will need to be incorporated like this (and Ben will have to update RPi.GPIO again)…

```
import RPi.GPIO as GPIO
if GPIO.RPI_REVISION == 1:
    ports = [0, 1, 21]
elif GPIO.RPI_REVISION == 2:
    ports = [2, 3, 27]
else:
    ports = ["whatever the new changes will be"]
print "Your Pi is a Revision %s, so your ports are: %s" % (GPIO.RPI_REVISION,
ports)
```

RPi.GPIO basics 3 – How to Exit GPIO programs cleanly, avoid warnings and protect your Pi

raspi.tv/2013/rpi-gpio-basics-3-how-to-exit-gpio-programs-cleanly-avoid-warnings-and-protect-your-pi

You might think I'm going about this series in a funny way. You'd be right! I am. That's because I'm trying to highlight the bits that people don't read about in other tutorials/documentation/manuals/books. These bits are useful, important and often overlooked.

If I covered inputs and outputs first, you might not get to the rest because you'd already "know all I need". That's only partially true though. Although you can get away with not knowing or using this stuff, it's much **better and safer if you know it *and use it***.

So today we're focussing on **how to exit cleanly** from a program that uses RPi.GPIO
I know – I'm bonkers! I haven't even told you how to **use** RPi.GPIO yet and I'm already showing you how to exit cleanly. Bear with me! It makes sense. **You need to know this stuff.**

Correct use of GPIO.cleanup()

RPi.GPIO provides a built-in function `GPIO.cleanup()` to clean up **all the ports you've used**. But be very clear what this does. **It only affects any ports you have set in the current program.** It resets any ports you have used in this program back to input mode. This prevents damage from, say, a situation where you have a port set HIGH as an output and you accidentally connect it to GND (LOW), which would short-circuit the port and possibly fry it. Inputs can handle either 0V (LOW) or 3.3V (HIGH), so it's safer to leave ports as inputs.

Here's a simple example of how to use `GPIO.cleanup()`…

```
import RPi.GPIO as GPIO

# the rest of your code would go here

# when your code ends, the last line before the program exits would
be...
GPIO.cleanup()

# remember, a program doesn't necessarily exit at the last line!
```

If you use this, and the program exits normally, all the ports you've used will be cleaned up and ready for use straight away. But life's not often as simple as that is it? More on that in a minute. First I want to show you…

Incorrect use of GPIO.cleanup()

I have seen people using `GPIO.cleanup()` wrongly at the start of a program, thinking it will clean up all the ports on the Pi. **It does no such thing.**[1] If you put it at the top of your script, it's just a wasted line of code. It does nothing until you've set up some ports for input or output (which we're covering soon).

Use `GPIO.cleanup()` on exit, as I've shown above, and in slightly more complex situations, as I will show below…

Reasons for a program exit

There are at least three reasons, that I can think of, for exiting a program…

1. **Natural, controlled end** – the program finished doing what it was written to do. Hurrah! It worked :)
2. **Error** – something went wrong and the program “throws a wobbly” and “errors out”. Either a process failed in an unforeseen way, or there was a coding error in the program.
3. **Keyboard Interrupt** – you pressed `CTRL+C` to stop the program. This is called a Keyboard Interrupt.

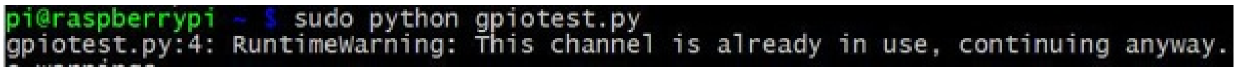
Both 2 & 3 are types of “exceptions”.

So what?

The problem is that, unless you code for these situations, any ports which are in use at the time of an error or Keyboard Interrupt will stay set exactly as they were, even after the program exits.

> *That’s an out of control situation – **unacceptable**, when you’re trying to take over the world with GPIO programming! Everything has to be under control.*

If you then try to run the program again, when you try to set up the ports that are “still set” you’ll get warnings that the ports are “already in use”.



```
pi@raspberrypi ~ $ sudo python gpiotest.py
gpiotest.py:4: RuntimeWarning: This channel is already in use, continuing anyway.
```

Switch off the warnings?

You can switch off the warnings, but that’s the “chicken’s way out”. It hides the problem, but doesn’t solve it. Still, you may choose to do it, so here’s how you do. Just place…

```
GPIO.setwarnings(False)
```

…near the top of your script, before you start setting up any ports.

Let’s try: a better way

My preferred method to catch errors and Keyboard Interrupts uses a **try: except:** block.
If you put your main piece of code in a **try:** block, it won’t “bomb straight out” of the program if an exception occurs (something doesn’t go according to plan).

You can specify different types of exceptions, including **KeyboardInterrupt**, and/or have a general one to catch every exception. There’s a list of Python Exceptions here. I’ve only ever used a couple of them.

Have a look at the code below to see how the exceptions are coded…

```
import RPi.GPIO as GPIO

# here you would put all your code for setting up GPIO,
# we'll cover that tomorrow
# initial values of variables etc...
counter = 0

try:
    # here you put your main loop or block of code
    while counter < 9000000:
        # count up to 9000000 - takes ~20s
        counter += 1
    print "Target reached: %d" % counter

except KeyboardInterrupt:
    # here you put any code you want to run before the
program
    # exits when you press CTRL+C
    print "\n", counter # print value of counter

except:
    # this catches ALL other exceptions including errors.
    # You won't get any error messages for debugging
    # so only use it once your code is working
    print "Other error or exception occurred!"

finally:
    GPIO.cleanup() # this ensures a clean exit
```

If you let the program run for ~22 seconds, it will count up to 9 million, tell you it reached its target, clean up any GPIO ports you've used and exit normally. This is the code within the `try:` block (lines 8-13).

The code in the `except KeyboardInterrupt:` block (lines 15-18) covers the CTRL+C situation. So when you press CTRL+C, it prints the current value of `counter`, cleans up any GPIO ports you've used, and exits.

The code in the `except:` block (lines 20-24) covers all other exceptions - including program errors. So if another exeption occurs, it warns you, cleans up any GPIO ports you've used, and exits.

This program will run until it finishes, or a keyboard interrupt or other exception occurs. Whichever of these occurs, the `finally:` block (lines 26-27) will run, cleaning up the GPIO ports on exit.

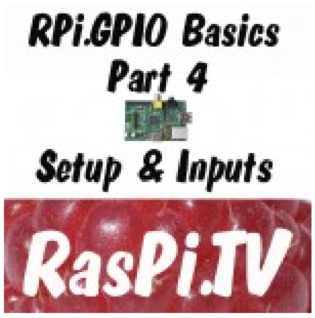
> *Also note, that we haven't actually used any GPIO ports here, so the cleanup isn't really doing anything yet, but this is a template you can use when we do start using ports.*

If you want to try it out, it's a bit long for a live Python session, so you could retype it, or cut and paste it into a program as we did in the first post in this series...

Now you know how to exit an RPi.GPIO program cleanly, without leaving a trail of uncontrolled ports behind you.

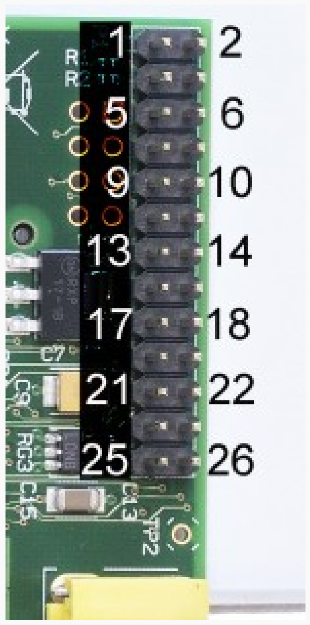
[1] If it did clean up all the ports, this would mean you could have major conflicts between different programs, which might not even be trying to use the same ports. Clearly, not a desirable situation!

RPi.GPIO basics 4 – Setting up RPi.GPIO, numbering systems and inputs



Well, today is the day we actually get to use RPi.GPIO a little bit. But, before we get to that, you should know about the two different numbering systems you can use with RPi.GPIO.

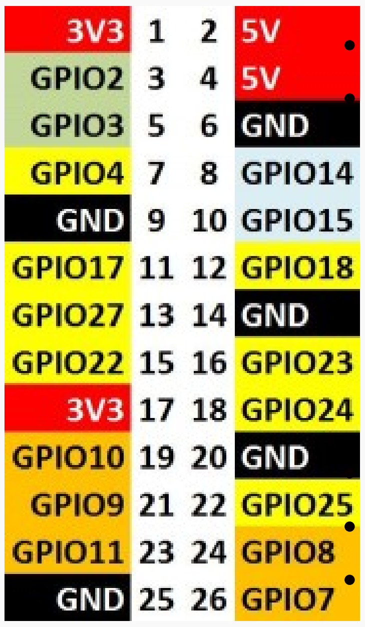
If you take a look at the main GPIO header (P1) of the Raspberry Pi, you'll see that there are 26 pins. The top left pin (as we look at this photo) is called pin 1, the one to the right of it is pin 2. So the next row is 3, 4 etc. and on down to 25, 26. This is how pin headers are numbered.



P1 Header

But Pins have names too

The slightly confusing part is that each pin also has a name, according to what it does. Some of them have alternative functions, but RPi.GPIO doesn't currently control those, so we'll ignore them for now. The best way to see which pin number does what is with a diagram…



3V3	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3V3	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

GPIO pinouts for Rev 2 Pi

- red ones are +ve power (3V3 or 5V)
- black ones are -ve ground
- yellow ones are all dedicated general purpose I/O ports (OK, 18 does PWM as well, but forget that for now).

The rest can all be used as GPIO ports, but do have other functions too. If you need 8 or less ports, it's best to use the yellow ones because you're less likely to have a conflict with other things. A quick rundown of what the others are…

- greeny/grey – i^{2}c interface
- light grey – UART (serial port)
- orange – SPI (Serial Peripheral Interface)

How to set up BOARD and GPIO numbering schemes

In RPi.GPIO you can use either pin numbers (BOARD) or the Broadcom GPIO numbers (BCM), **but you can only use one system in each program.** I habitually use the GPIO numbers, but neither way is wrong. Both have advantages and disadvantages.

> *If you use pin numbers, you don't have to bother about revision checking, as RPi.GPIO takes care of that for you. You still need to be aware of which pins you can and can't use though, since some are power and GND.*
>
> *If you use GPIO numbers, your scripts will make better sense if you use a Gertboard, which also*

> *uses GPIO numbering. If you want to use the P5 header for GPIO28-31, you have to use GPIO numbering. If you want to control the LED on a Pi camera board (GPIO5) you also have to use GPIO numbering.*

The important thing is to pick the one that makes sense to you and use it. It's also important to be aware of the other system in case you ever need to work on someone elses code.

So, at the top of every script, after importing the RPi.GPIO module, we set our GPIO numbering mode.

```
import RPi.GPIO as GPIO

# for GPIO numbering, choose BCM
GPIO.setmode(GPIO.BCM)

# or, for pin numbering, choose BOARD
GPIO.setmode(GPIO.BOARD)

# but you can't have both, so only use
one!!!
```

So, with a drumroll and a fanfare of trumpets, it's now time for us to set up some inputs.

How to set up a GPIO port as an input

Use the following line of code…

```
GPIO.setup(Port_or_pin,
GPIO.IN)
```

…changing *Port_or_pin* to the number of the GPIO port or pin you want to use. I'm going to use the BCM GPIO numbering and port GPIO25, so it becomes…

```
GPIO.setup(25,
GPIO.IN)
```

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)  # set up BCM GPIO
numbering
GPIO.setup(25, GPIO.IN) # set GPIO 25 as input
```

We've just set up port 25 as an input. Next we need to be able to "read" the input.

Reading inputs

Inputs are Boolean values: 1 or 0, GPIO.HIGH or GPIO.LOW, True or False (this corresponds to the voltage on the port: 0V=0 or 3.3V=1). You can read the value of a port with this code…

```
GPIO.input(25)
```

But it may be more useful to use it as part of your logic…

```
if GPIO.input(25): # if port 25 == 1
    print "Port 25 is
1/GPIO.HIGH/True"
```

…or store its value in a variable to use in a different part of the program…

```
button_press =
GPIO.input(25)
```

So, building on what we've already done, here's a very simple program to read and display the status of port 25, but it only does it once, then it cleans up and exits.

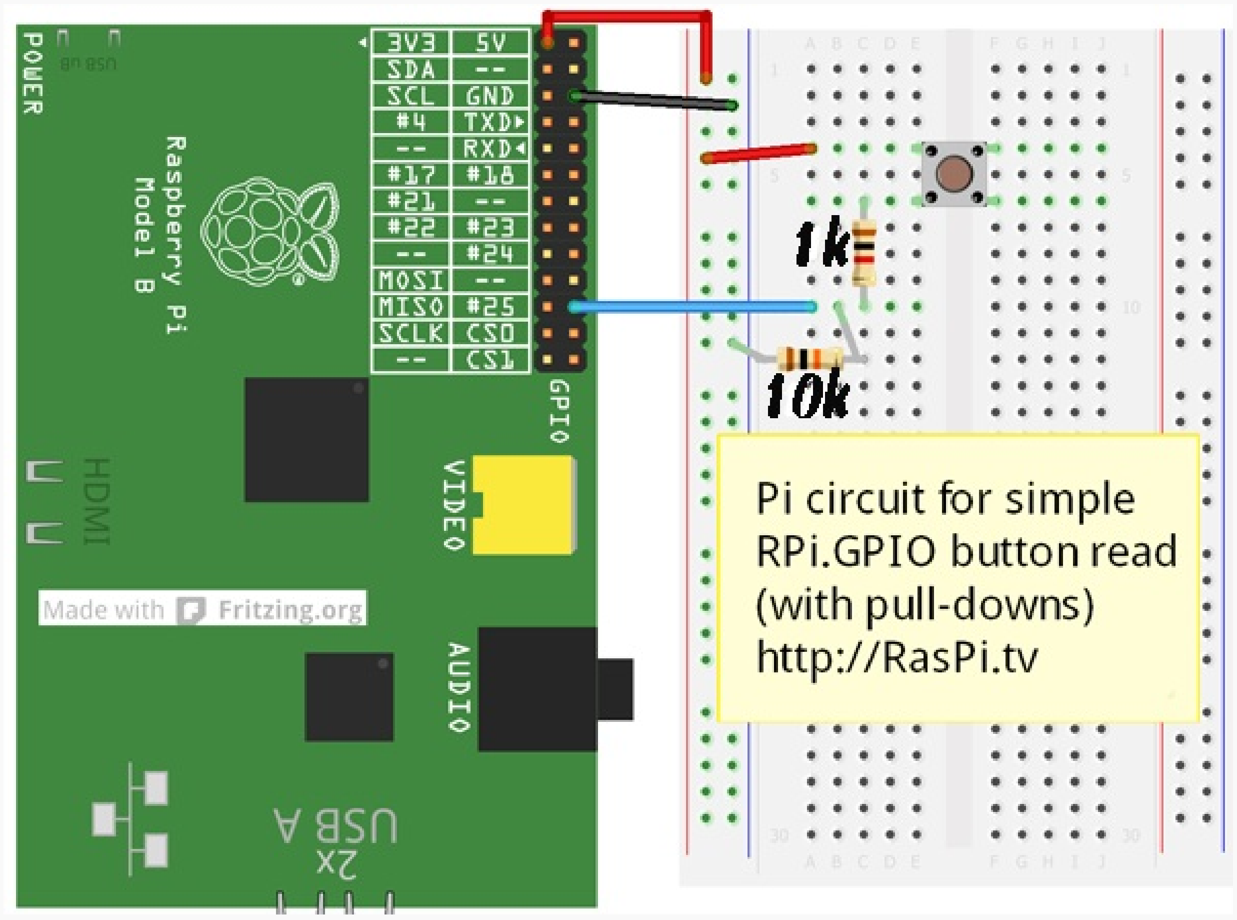
```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)          # set up BCM GPIO
numbering
GPIO.setup(25, GPIO.IN)         # set GPIO 25 as input

if GPIO.input(25):              # if port 25 == 1
    print "Port 25 is 1/GPIO.HIGH/True"
else:
    print "Port 25 is 0/GPIO.LOW/False"

GPIO.cleanup()                  # clean up after yourself
```

The above is not really complete yet, what we need to do is add…

- a loop, so we can make it read more than once
- a time delay, so it won't read the port thousands of times per second
- a circuit with a button, so we can change the status of the port and see the input status change on the screen
- a **try: except KeyboardInterrupt:** block so we can exit cleanly (we covered this yesterday)

So, let's make a little circuit

In order to go any further with this, we need to make up a little circuit.



Circuit with button for manipulating GPIO25 (click to enlarge)

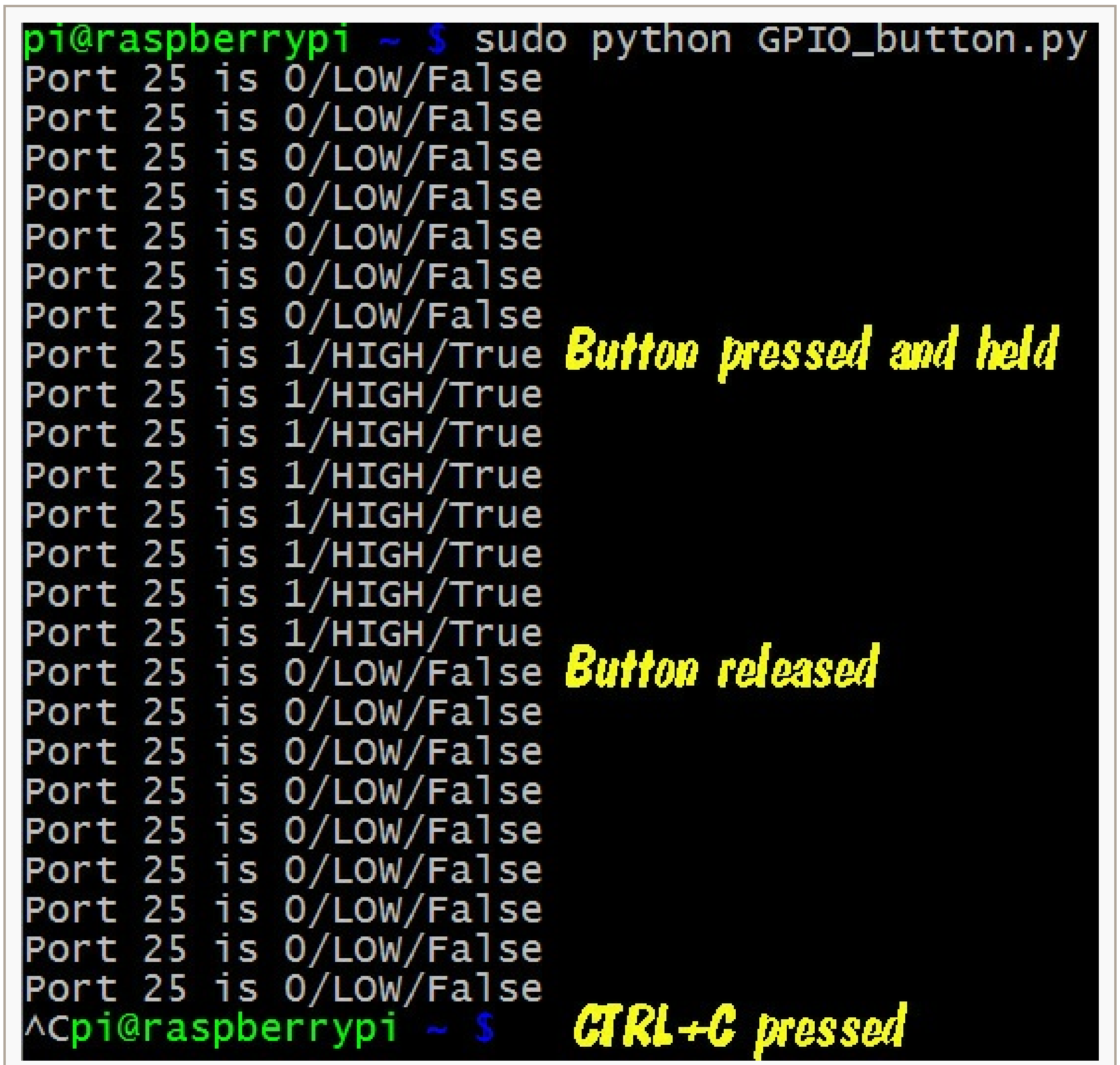
The resistors are used to “pull down” the GPIO25 pin to 0 Volts GND (0V, 0, LOW, False) unless the button is pressed. When the button is pressed, GPIO25 is connected to 3.3V. If you don’t use the resistors, it might still work, but you risk having a “floating” port. It’s not dangerous, (it could be if you had something like a motor attached) but it’s not fully in control either. We’ll cover a bit more on floating ports, pull-ups and pull-downs in another article (probably part 6).

> *If you don’t have a breadboard, resistors and buttons, you could cheat and just use a jumper wire to connect two pins directly, but you’ll need to be aware of what you’re doing.*

Here’s the code with all the extras added to read and display the button’s status, and exit cleanly when CTRL+C is pressed…

```
import RPi.GPIO as GPIO
from time import sleep     # this lets us have a time delay (see line
12)
GPIO.setmode(GPIO.BCM)     # set up BCM GPIO numbering
GPIO.setup(25, GPIO.IN)    # set GPIO 25 as input

try:
    while True:            # this will carry on until you hit CTRL+C
        if GPIO.input(25): # if port 25 == 1
            print "Port 25 is 1/GPIO.HIGH/True - button pressed"
        else:
            print "Port 25 is 0/GPIO.LOW/False - button not pressed"
        sleep(0.1)         # wait 0.1 seconds

except KeyboardInterrupt:
    GPIO.cleanup()         # clean up after yourself
```

This is what the above program's output looks like…



```
pi@raspberrypi ~ $ sudo python GPIO_button.py
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 1/HIGH/True
Port 25 is 1/HIGH/True
Port 25 is 1/HIGH/True
Port 25 is 1/HIGH/True
Port 25 is 1/HIGH/True
Port 25 is 1/HIGH/True
Port 25 is 1/HIGH/True
Port 25 is 1/HIGH/True
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
Port 25 is 0/LOW/False
^Cpi@raspberrypi ~ $
```

Output from button press program

Polling loop

This program uses a polling loop. It checks (polls) the input port 10 times per second. It works well, but is not very efficient. A more efficient, and more advanced way to handle this is with interrupts. I've written about how you can use interrupts in RPi.GPIO here.

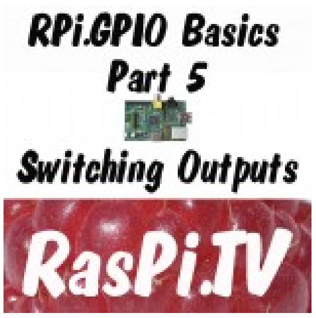
That was inputs with RPi.GPIO

So now you know how to use RPi.GPIO to set up and read the status of an input port. In part 5, we'll cover Outputs.

RPi.GPIO basics 5 – Setting up and using outputs with RPi.GPIO

raspi.tv/2013/rpi-gpio-basics-5-setting-up-and-using-outputs-with-rpi-gpio

Today is output day. I'm going to show you how to switch things on and off using RPi.GPIO to control the output ports of the Raspberry Pi.



Once you can control outputs, you can, with a few additional electronic components, switch virtually anything on and off. Given the Raspberry Pi's excellent connectivity, this means you can switch things on and off through the internet, using any computer, smartphone or tablet from anywhere in the world. You can also use a local network, Bluetooth etc. for short-range control.

Setting up is very similar to the way we set things up for inputs, but instead of…

```
GPIO.setup(port_or_pin,
GPIO.IN)
```

we use…

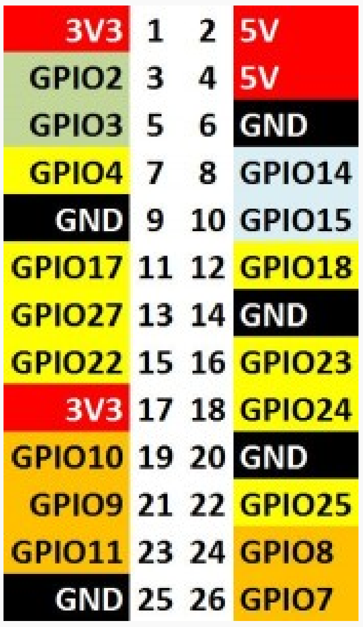
```
GPIO.setup(port_or_pin, GPIO.OUT)
```

Then, to switch the port/pin to 3.3V (equals 1/GPIO.HIGH/True)…

```
GPIO.output(port_or_pin,
1)
```

Or, to switch the port/pin to 0V (equals 0/GPIO.LOW/False)…

```
GPIO.output(port_or_pin,
0)
```



3V3	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3V3	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

P1

How to set an output – full Python code

```
import RPi.GPIO as GPIO            # import RPi.GPIO module
GPIO.setmode(GPIO.BCM)             # choose BCM or BOARD
GPIO.setup(port_or_pin, GPIO.OUT) # set a port/pin as an output
GPIO.output(port_or_pin, 1)        # set port/pin value to
1/GPIO.HIGH/True
GPIO.output(port_or_pin, 0)        # set port/pin value to
0/GPIO.LOW/False
```

You can also set the initial value of the output at the time of setting up the port with `initial=x` optional extra argument…

```
GPIO.setup(port_or_pin, GPIO.OUT, initial=1)
```
or
```
GPIO.setup(port_or_pin, GPIO.OUT, initial=0)
```

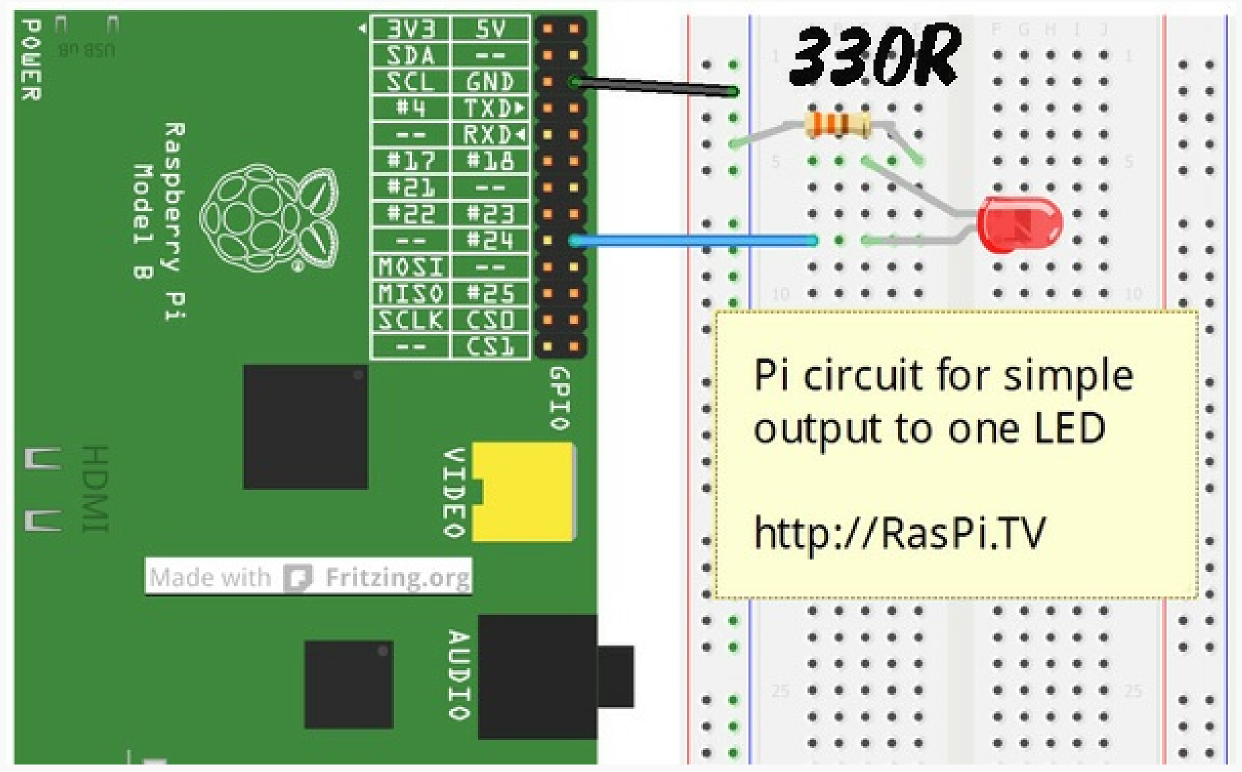
And that's really (almost) all there is to it. You can use GPIO.HIGH or GPIO.LOW and True or False as well, but I prefer 1 or 0 because it's less keystrokes.

Working example

Let's have a working example. We'll set up RPi.GPIO in BCM mode, set GPIO24 as an output, and switch it on and off every half second until we CTRL+C exit.

```
import RPi.GPIO as GPIO            # import RPi.GPIO module
from time import sleep             # lets us have a delay
GPIO.setmode(GPIO.BCM)             # choose BCM or BOARD
GPIO.setup(24, GPIO.OUT)           # set GPIO24 as an output

try:
    while True:
        GPIO.output(24, 1)         # set GPIO24 to 1/GPIO.HIGH/True
        sleep(0.5)                 # wait half a second
        GPIO.output(24, 0)         # set GPIO24 to 0/GPIO.LOW/False
        sleep(0.5)                 # wait half a second

except KeyboardInterrupt:          # trap a CTRL+C keyboard interrupt
    GPIO.cleanup()                 # resets all GPIO ports used by this
program
```

You can check this is working either by connecting a Voltmeter/oscilloscope to port 24 and GND or by using this circuit with a resistor (330R) and LED, which will flash on and off every half second. When you press CTRL+C to exit, the GPIO ports we used are reset.



Circuit for single LED output on GPIO24

Limit the current on the 3V3 supply

All the GPIO ports take their power from the Raspberry Pi's 3.3V (3V3) supply. The maximum recommended current draw from that supply is 51 mA. This is the **total for all the 3V3 GPIO pins** .

This is more than enough for controlling logic gates and integrated circuits. But if you are running LEDs directly from the ports, **it is possible to overload them**. You need to be careful about this.

The **maximum current draw from any one pin should not exceed 16 mA** . Many LEDs can draw more current than this, so you need to choose a resistor value that limits the current to an acceptably safe value. (One of my favourite LED calculators can be found here.)

I chose 330 &#8486 (330R) for this example because it should be high enough to be safe for pretty much any LED you could choose. The LED might not be as bright as it would with a lower value resistor, but your Pi ports should be OK.

> *Generally speaking, you should use the GPIO ports to trigger/switch things rather than to power things. We'll cover the use of a Darlington array chip, which helps you do this, in a later article.*

How to read the status of an output pin

Sometimes, when using a port as an output, it might be useful to be able to "read" its status – whether it's 1/GPIO.HIGH/True or 0/GPIO.LOW/False.

You can do this by treating it as an input port.
`GPIO.input(24)`

Let's modify the LED script to read the status of GPIO24 (lines 10-11 & 14-15) and give us a message about the LED's status. (This is a very simple and controlled example. You wouldn't normally use it in this situation – I'm just showing you how.)

```
import RPi.GPIO as GPIO           # import RPi.GPIO module
from time import sleep            # lets us have a delay
GPIO.setmode(GPIO.BCM)            # choose BCM or BOARD
GPIO.setup(24, GPIO.OUT)          # set GPIO24 as an output

try:
    while True:
        GPIO.output(24, 1)        # set GPIO24 to 1/GPIO.HIGH/True
        sleep(0.5)                # wait half a second
        if GPIO.input(24):
            print "LED just about to switch off"
        GPIO.output(24, 0)        # set GPIO24 to 0/GPIO.LOW/False
        sleep(0.5)                # wait half a second
        if not GPIO.input(24):
            print "LED just about to switch on"
except KeyboardInterrupt:         # trap a CTRL+C keyboard interrupt
    GPIO.cleanup()                # resets all GPIO ports used by this
program
```

So now you know how to use RPi.GPIO to set up and control outputs on the Raspberry Pi.

RPi.GPIO basics 6 – Using inputs and outputs together with RPi.GPIO – pull-ups and pull-downs

raspi.tv/2013/rpi-gpio-basics-6-using-inputs-and-outputs-together-with-rpi-gpio-pull-ups-and-pull-downs

Today, it's time for us to combine inputs and outputs into the same script. It's nothing scary and it's not new either. It's simply a case of doing what we've already done in the last two days' of GPIO basics, but combining them.

To make it super-easy, we'll even stick to the same port numbers we used for the last two days. We're going to make a new program which takes parts from both the "read a button press" and the "flash an led every half second" programs.

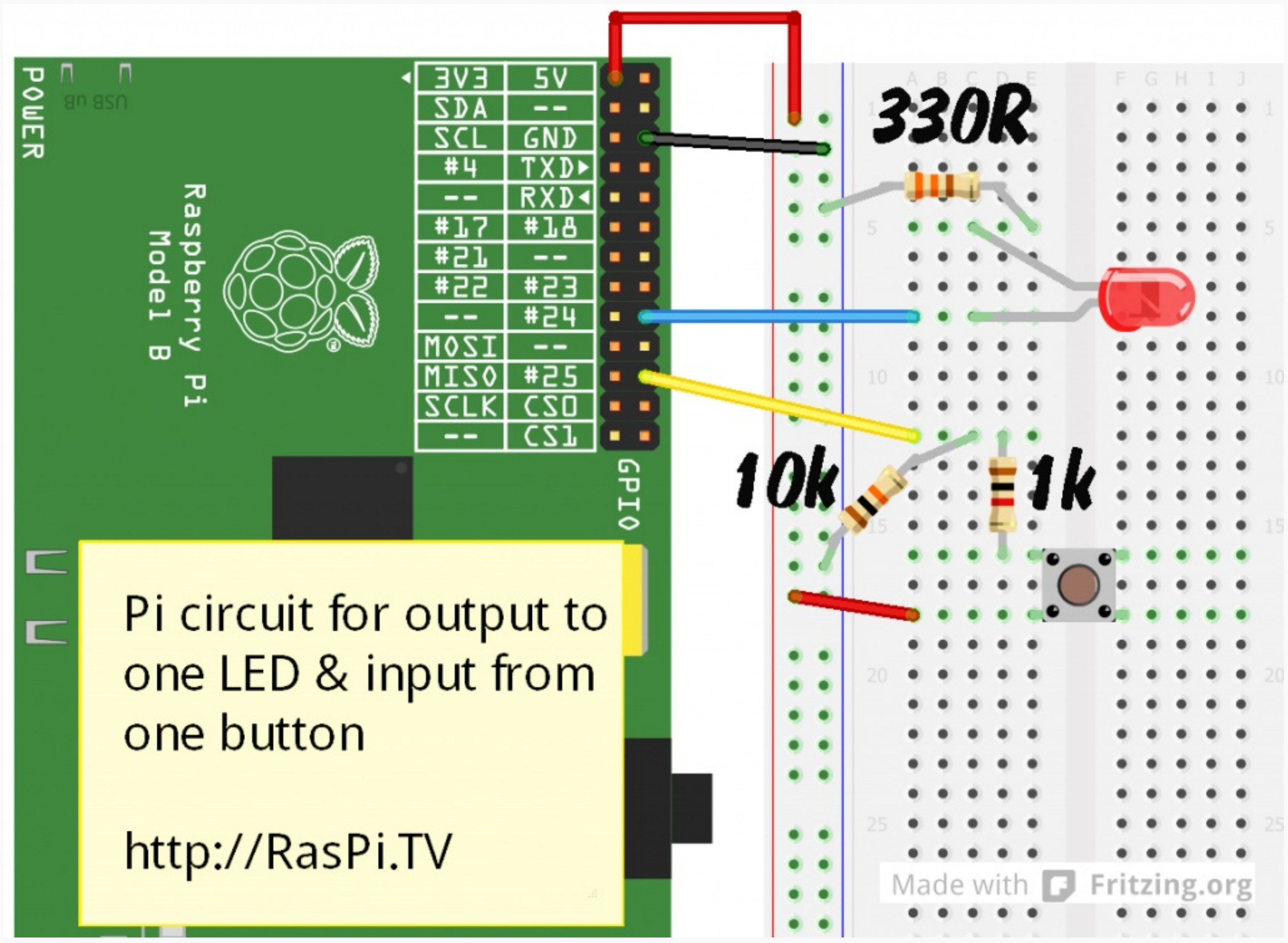
We're going to make a simple new program which switches the LED on when the button is pressed and switches it off again when the button is released. But before we do that…

Here's a quick recap of inputs and outputs with RPi.GPIO

```
import RPi.GPIO as GPIO           # import RPi.GPIO module
GPIO.setmode(GPIO.BCM)            # choose BCM or BOARD
GPIO.setup(port_or_pin, GPIO.IN)  # set a port/pin as an input
GPIO.setup(port_or_pin, GPIO.OUT) # set a port/pin as an output
GPIO.output(port_or_pin, 1)       # set an output port/pin value to 1/HIGH/True
GPIO.output(port_or_pin, 0)       # set an output port/pin value to 0/LOW/False
i = GPIO.input(port_or_pin)       # read status of pin/port and assign to
variable i
```

So now we need a combined circuit

This is simply a combination of the circuits from the last two exercises.



Circuit for combining button input and LED output with external pull-down resistor

Here's a simple Python program

This takes bits from each of the previous two programs.

```
import RPi.GPIO as GPIO
from time import sleep      # this lets us have a time delay (see line 15)
GPIO.setmode(GPIO.BCM)      # set up BCM GPIO numbering
GPIO.setup(25, GPIO.IN)     # set GPIO25 as input (button)
GPIO.setup(24, GPIO.OUT)    # set GPIO24 as an output (LED)

try:
    while True:             # this will carry on until you hit CTRL+C
        if GPIO.input(25): # if port 25 == 1
            print "Port 25 is 1/HIGH/True - LED ON"
            GPIO.output(24, 1)         # set port/pin value to 1/HIGH/True
        else:
            print "Port 25 is 0/LOW/False - LED OFF"
            GPIO.output(24, 0)         # set port/pin value to 0/LOW/False
        sleep(0.1)          # wait 0.1 seconds

finally:                    # this block will run no matter how the try block
exits
    GPIO.cleanup()          # clean up after yourself
```

Every 0.1s, this program checks the button status…

- if pressed (input port 25 == 1), button status is displayed and the LED is switched on (output port 24 is set to 1)
- otherwise, if not pressed (input port 25 == 0), button status is displayed and the LED is switched off (output port 24 is set to 0)
- It keeps going until CTRL+C is pressed, then the ports are cleaned up before exit

Now YOU make it better

That's the input/output (I/O) part sorted, now it's over to you to improve the script. I can think of several improvements you could try to make.

- **Fewer messages:** Only display a message when the button status changes, rather than on every iteration of the loop.
- **Light switch mode:** Make it switch on the LED when the button is pressed and released once and leave it on until the button is pressed and released again. (You might need to investigate a little bit about "button debounce" if you run into difficulties)
- **Flashing:** Reinstate the flashing, while keeping the "on – off" light switch mode

Internal pull-ups and pull-downs

In the button circuit, we're using resistors to "pull-down" the port. This gives it a "default" state of 0V (0, LOW, False). I mentioned in day 5 that we'd cover this in more detail, so that's what we're doing now.

Why are they necessary/useful?

If you have no pull-up or pull-down resistors attached to an input port, its status is not clearly defined. It is "floating". It is susceptible to random electromagnetic radiation or static from you, from any devices near or far and from the environment. Any wires attached to the GPIO ports act as antennae for this radiation (it's mostly radio waves).

So imagine the situation where you have a motor with a propellor on it, or one which controls Granny's stairlift, which is controlled by a GPIO input port. If that port is susceptible to random changes of state, the propellor might spin when it shouldn't and hurt someone. Or Granny might be sent back upstairs when she wants to be downstairs. It's an out-of-control situation. You can easily get it under control by using pull-up or pull-down resistors, so that's what we do.

What's all this about internal ones then?

In our button circuit, we used resistors to pull down the voltage. This was to demonstrate the idea.

But, in fact, the Raspberry Pi has built-in pull-up and pull-down resistors which can be enabled in software. This means we can eliminate our pull-down resistors for the button – as long as we enable the internal ones.

How do you do that? RPi.GPIO to the rescue.

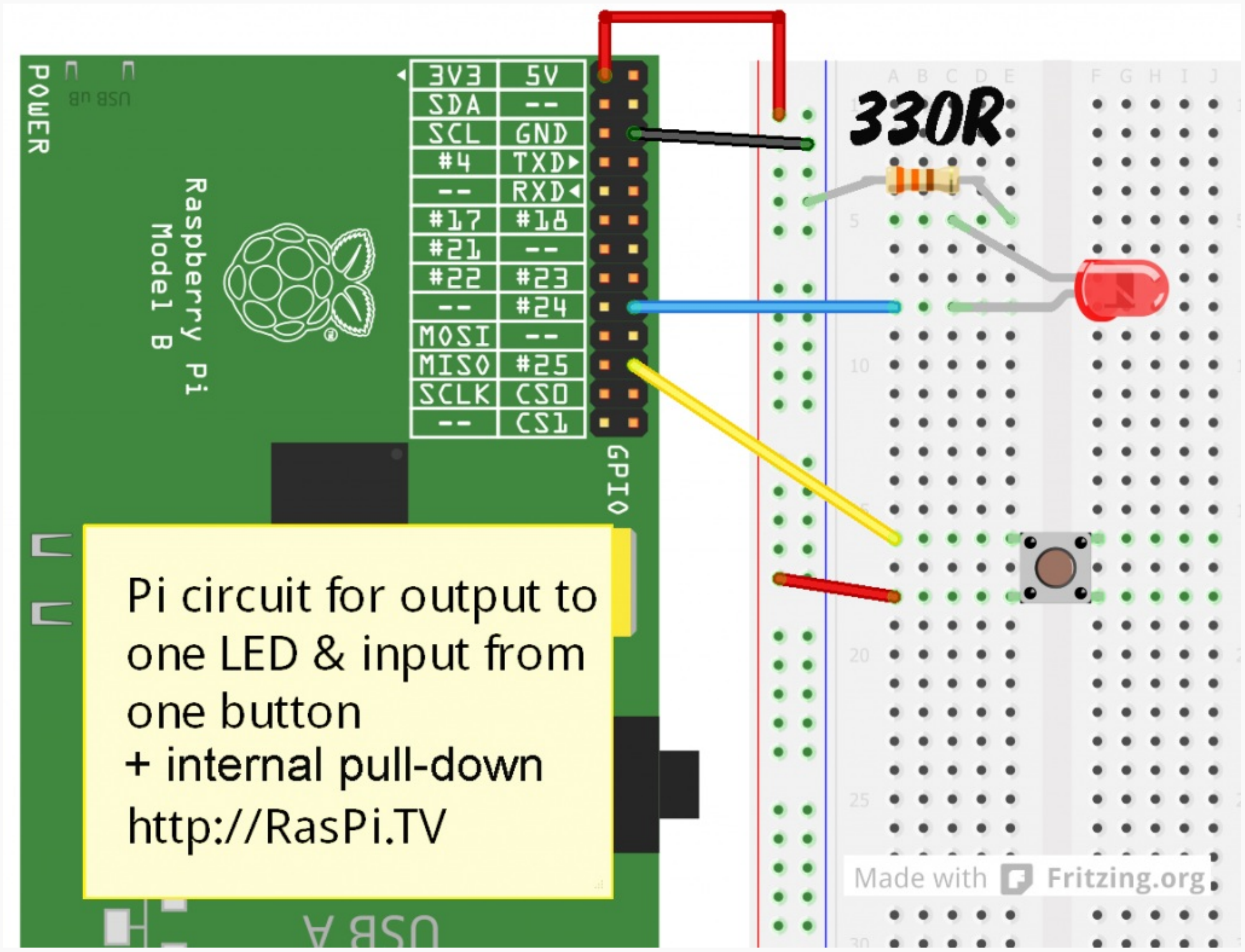
You enable these internal pull-ups/pull-downs at the time of setting up the port for input or output, by adding an extra, optional, argument to the GPIO.setup() function call.

We're using pull-down, so it's **pull_up_down=GPIO.PUD_DOWN**. If you want/need pull-up you can change the PUD_DOWN for PUD_UP. (It depends on how you've wired your circuit.)

```
GPIO.setup(port_or_pin, GPIO.IN,
pull_up_down=GPIO.PUD_DOWN)
```

How does that apply to our program?

We'll need to change the circuit first, to this (you could also just remove the 10k resistor to GND from the previous circuit)…



Circuit for combining button input and LED output using internal pull-down resistor

…and then, once you've done that, line 4 needs changing to enable the pull-down on the button port (25). Change line 4 from…

```
GPIO.setup(25, GPIO.IN) # set GPIO25 as input
(button)
```

…to…

```
GPIO.setup(25, GPIO.IN, pull_up_down=GPIO.PUD_DOWN) # set GPIO25 as input
(button)
```

The whole program should then look like this…

```
import RPi.GPIO as GPIO
from time import sleep     # this lets us have a time delay (see line 15)
GPIO.setmode(GPIO.BCM)     # set up BCM GPIO numbering
GPIO.setup(25, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)    # set GPIO25 as input
(button)
GPIO.setup(24, GPIO.OUT)   # set GPIO24 as an output (LED)

try:
    while True:            # this will carry on until you hit CTRL+C
        if GPIO.input(25): # if port 25 == 1
            print "Port 25 is 1/HIGH/True - LED ON"
            GPIO.output(24, 1)         # set port/pin value to 1/HIGH/True
        else:
            print "Port 25 is 0/LOW/False - LED OFF"
            GPIO.output(24, 0)         # set port/pin value to 0/LOW/False
        sleep(0.1)         # wait 0.1 seconds

finally:                   # this block will run no matter how the try block exits
    GPIO.cleanup()         # clean up after yourself
```

As you'll see if you try it out, it works just the same as the previous version.

That was simultaneous inputs and outputs in RPi.GPIO, along with internal pull-ups/pull-downs

So now you know how to use inputs and outputs at the same time with RPi.GPIO in Python on the Raspberry Pi. You also, hopefully understand a bit about pull-up and pull-down resistors and why they are used.

RPi.GPIO basics 7 – RPi.GPIO cheat sheet and pointers to RPi GPIO advanced tutorials

raspi.tv/2013/rpi-gpio-basics-7-rpi-gpio-cheat-sheet-and-pointers-to-rpi-gpio-advanced-tutorials

In the previous six articles, we've covered the basics of RPi.GPIO. I hope the series has been useful.

While I was doing the "recap" for part 6, I decided to make myself a quick reference "cheat sheet" with all the common RPi.GPIO stuff in it. I figured this would be useful and save me having to look things up on the web and in previous Python scripts I've written. I made it as a text file that I can just copy and paste snippets of code from to make RPi.GPIO Python coding easier.

Then I realised that it would make a perfect part 7 to round off the series – along with links to the more advanced RPi.GPIO tutorials on interrupts and PWM that I published back in April. So here it is…

RasPi.TV RPi.GPIO Quick Reference "cheat sheet"

I'll put it here as a "Python script", and there's also a downloadable PDF with a full breakdown of all the P1 ports for both Rev 1 and Rev 2 Pis and a full list of links to all 12 RPi.GPIO tutorials on RasPi.TV.

There's also a .txt version you can either download or wget straight onto your Pi.

```
wget
http://RasPi.TV/download/rpigpio.txt
```

```
# RPi.GPIO Basics cheat sheet - Don't try to run this. It'll fail!
# Alex Eames http://RasPi.TV
# http://RasPi.TV/?p=4320

# RPi.GPIO Official Documentation
# http://sourceforge.net/p/raspberry-gpio-python/wiki/Home/

import RPi.GPIO as GPIO              # import RPi.GPIO module

# choose BOARD or BCM
GPIO.setmode(GPIO.BCM)               # BCM for GPIO numbering
GPIO.setmode(GPIO.BOARD)             # BOARD for P1 pin numbering

# Set up Inputs
GPIO.setup(port_or_pin, GPIO.IN)     # set port/pin as an input
GPIO.setup(port_or_pin, GPIO.IN,  pull_up_down=GPIO.PUD_DOWN) # input with pull-
down
GPIO.setup(port_or_pin, GPIO.IN,  pull_up_down=GPIO.PUD_UP)   # input with pull-up

# Set up Outputs
GPIO.setup(port_or_pin, GPIO.OUT)               # set port/pin as an output
GPIO.setup(port_or_pin, GPIO.OUT, initial=1)    # set initial value option (1 or
0)

# Switch Outputs
GPIO.output(port_or_pin, 1)     # set an output port/pin value to 1/GPIO.HIGH/True
GPIO.output(port_or_pin, 0)     # set an output port/pin value to 0/GPIO.LOW/False

# Read status of inputs OR outputs
i = GPIO.input(port_or_pin)     # read status of pin/port and assign to variable i
if GPIO.input(port_or_pin):     # use input status directly in program logic

# Clean up on exit
GPIO.cleanup()

# What Raspberry Pi revision are we running?
GPIO.RPI_REVISION    #  0 = Compute Module, 1 = Rev 1, 2 = Rev 2, 3 = Model B+

# What version of RPi.GPIO are we running?
GPIO.VERSION

# What Python version are we running?
import sys; sys.version
```

Links to more advanced RPi.GPIO tutorials

If this RPi.GPIO basics series has made you want to dig deeper into GPIO programming/physical computing, you'll be glad to hear that there are some more advanced features available in RPi.GPIO.

Interrupts are a much more efficient way of handling "button press" type inputs than the "polling loop" we used in our basic example. There's a series of three articles about interrupts.

Pulse-width Modulation (PWM) can be used to control things (e.g. motor speed) more precisely than just on and off. There's a two-part series on that as well.

Interrupts (needs RPi.GPIO 0.5.2+)

- **Background and simple interrupt:** How to use interrupts with Python on the Raspberry Pi and RPi.GPIO
- **Threaded callback:** How to use interrupts with Python on the Raspberry Pi and RPi.GPIO – part 2
- **Multiple threaded callback:** How to use interrupts with Python on the Raspberry Pi and RPi.GPIO – part 3
- **Edge Detection:** Detecting both rising and falling edges with RPi.GPIO

Software PWM

From 0.5.2a onwards, RPi.GPIO has software PWM. Below are two links. One describes its basic usage and the other is a practical application for dimming LEDs and controlling motor speed.

- RPi.GPIO 0.5.2a now has software PWM – How to use it
- How to use soft PWM in RPi.GPIO 0.5.2a pt 2 – led dimming and motor speed control

I haven't forgotten about the Darlington

In part 5, I mentioned a future article about the use of a Darlington array. That's not strictly RPi.GPIO, and will come at some point, but if you're desperately impatient about that you can click here to see a bit about Darlington arrays as part of the PWM tutorial.

Thanks Ben

No series on RPi.GPIO would be complete without a big thank you to RPi.GPIO author Ben Croston. Thank you for all your hard work on making it easy for us to control the world with GPIO on our Raspberry Pis. The official RPi.GPIO documentation is found here.

Download the cheat sheet

So, that's about it for RPi.GPIO basics.

You can download the Quick Reference cheat sheet here.

And you can download the 'cut and paste' text version here. or
```
wget http://RasPi.TV/download/rpigpio.txt
```

Links to all the RPi.GPIO Basics series

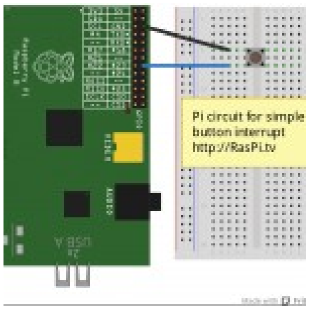
1. How to check what RPi.GPIO version you have
2. How to check what Pi board Revision you have
3. How to Exit GPIO programs cleanly, avoid warnings and protect your Pi
4. Setting up RPi.GPIO, numbering systems and inputs
5. Setting up and using outputs with RPi.GPIO
6. Using inputs and outputs at the same time with RPi.GPIO, and pull-ups/pull-downs
7. RPi.GPIO cheat sheet

I hope these will be really useful to you. It's been an intense week putting it all together. Have fun taking over the world, one GPIO port at a time.

How to use interrupts with Python on the Raspberry Pi and RPi.GPIO

raspi.tv /2013/how-to-use-interrupts-with-python-on-the-raspberry-pi-and-rpi-gpio

The latest big news in the world of Raspberry Pi Python GPIO programming is that Ben Croston has released an update for RPi.GPIO. Why is that a big deal? Because this version has interrupts. "What's an interrupt?" I hear you say. It's a way of waiting for something to happen without checking constantly whether or not it's happening.



Imagine that you're waiting for a delivery – something you're really excited about – like a Pi camera.You spend far too much time looking down the street in eager anticipation of the postman's arrival. You can't fully focus on what you're supposed to be doing because you know it's imminent. Every time you hear something in the street, you jump up and look out of the window. Woe betide any door-knocking salesman who calls when you're expecting a delivery.

What I've just described in human terms is a bit like polling. Polling is continually checking for something. For example, if you want to make sure your program reacts as quickly as possible to a button press, you can check the button status about ten thousand times per second. This is great if you need a quick reaction, but it uses quite a bit of the computer's processing power. In the same way that you can't fully focus when you're expecting a delivery, a large part of your CPU is used up with this polling.

There has to be a better way, right?

Yes. And there is. It's interrupts. This is the first in a series of articles which aim to show you how to use this new interrupt facility in Python.

Interrupts are a much more efficient way of handling the "wait for something to happen and react immediately when it does" situation. They free up the resources you would have wasted on polling, so that you can use them for something else. Then, when the event happens, the rest of your program is "interrupted" and your chosen outcome occurs.

So, to carry on our human example…
An interrupt is like having an automatic postman detector that will tell you for sure when the postman arrives, so you can get on with something else. You now know you will not miss that knock on the door and end up with one of those "we tried to deliver your item but you were out and the collection office is closed for the next two days, so enjoy the wait" cards.

So interrupts are good, as you can set them up to wait for events to occur without wasting system resources.

So how do you code them?

I'm going to show a simple "wait for a button press" example in this blog article and follow up with other examples in subsequent articles. But before you try this, you will quite likely need to update your RPi.GPIO package. You can check what version of RPi.GPIO you have in the command line with…

```
sudo python
import RPi.GPIO as
GPIO
GPIO.VERSION
```

This should show you what RPi.GPIO version you have. You need 0.5.1 or higher for this example.
You can exit the python environment with `CTRL+Z`

Install RPi.GPIO version 0.5.1 for simple interrupts

If you need to, you can install 0.5.1 or later with

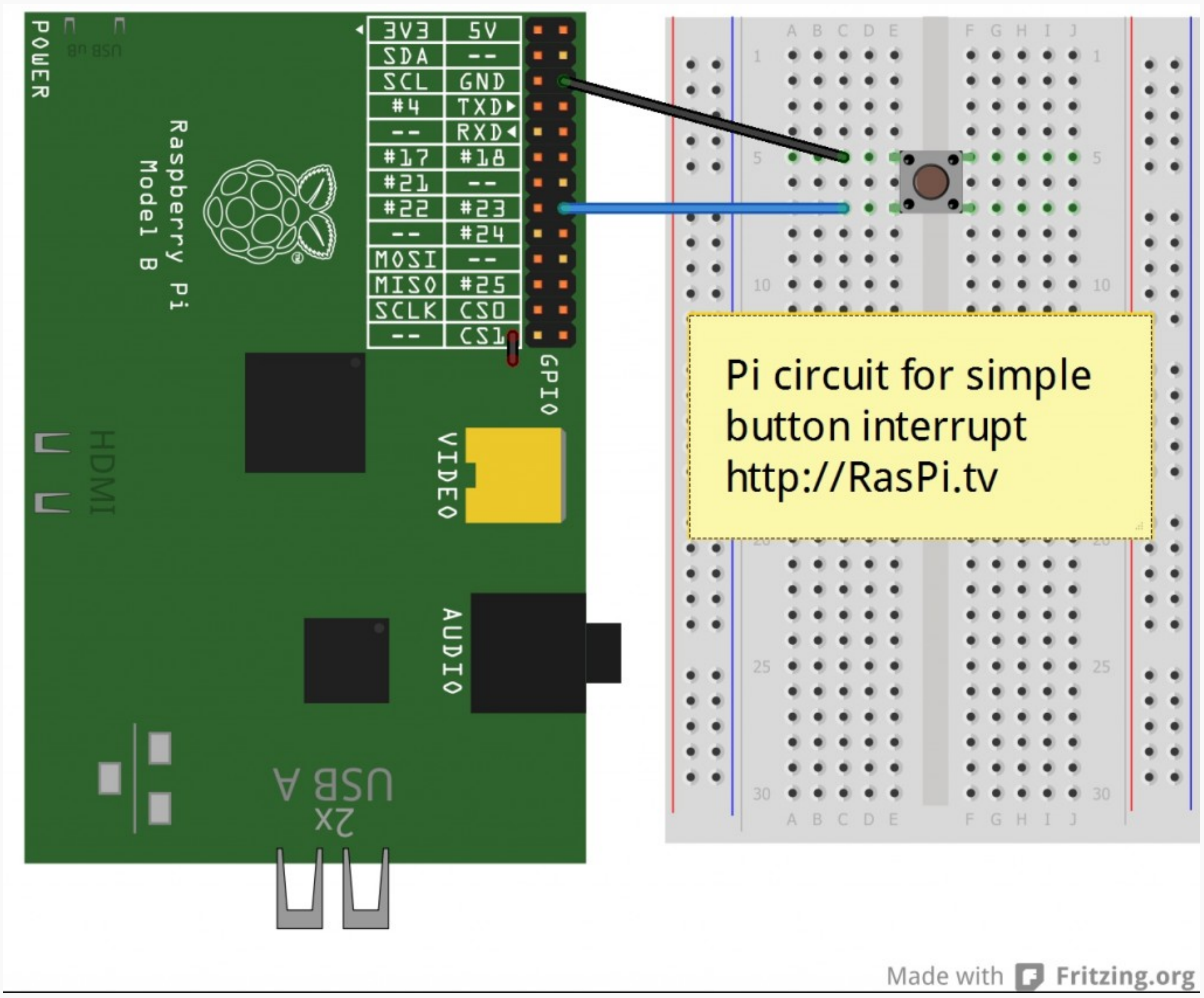
```
sudo apt-get update
sudo apt-get dist-
upgrade
```

(This will update all your Raspbian packages and may take up to an hour)

or, from the command line prompt (this will only update RPi.GPIO)…

```
wget http://raspberry-gpio-python.googlecode.com/files/python-rpi.gpio_0.5.1a-
1_armhf.deb
wget http://raspberry-gpio-python.googlecode.com/files/python3-rpi.gpio_0.5.1a-
1_armhf.deb
sudo dpkg -i python-rpi.gpio_0.5.1a-1_armhf.deb
sudo dpkg -i python3-rpi.gpio_0.5.1a-1_armhf.deb
```

And now the circuit



Circuit for simple button press interrupt

It’s simply a question of rigging up a button connecting 23 to GND when pressed.

And now onto the code

I’ve put most of the explanation in the code, so that if you use it, you will still have it.

```
#!/usr/bin/env python2.7
# script by Alex Eames http://RasPi.tv/
# http://raspi.tv/2013/how-to-use-interrupts-with-python-on-the-raspberry-pi-and-
rpi-gpio
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

# GPIO 23 set up as input. It is pulled up to stop false signals
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP)

print "Make sure you have a button connected so that when pressed"
print "it will connect GPIO port 23 (pin 16) to GND (pin 6)\n"
raw_input("Press Enter when ready\n>")

print "Waiting for falling edge on port 23"
# now the program will do nothing until the signal on port 23
# starts to fall towards zero. This is why we used the pullup
# to keep the signal high and prevent a false interrupt

print "During this waiting time, your computer is not"
print "wasting resources by polling for a button press.\n"
print "Press your button when ready to initiate a falling edge interrupt."
try:
    GPIO.wait_for_edge(23, GPIO.FALLING)
    print "\nFalling edge detected. Now your program can continue with"
    print "whatever was waiting for a button press."
except KeyboardInterrupt:
    GPIO.cleanup()       # clean up GPIO on CTRL+C exit
GPIO.cleanup()           # clean up GPIO on normal exit
```

Decoding the code

lines 4-5 import the RPi.GPIO module and set up the BCM port numbering scheme

line 8 sets GPIO 23 as an input with the pullup resistor set to UP.
This means that the signal will be HIGH all the time until the button is pressed connecting the port to GND, which makes it LOW. This avoids false event detection.

lines 10-11 print some instructions
line 12 waits for user to hit enter before starting. This gives an opportunity to check the wiring

lines 14-21 further instructions and documentation

line 22 `try:` & line 26 `except KeyboardInterrupt:`
This allows us to run the program and exit cleanly if someone presses `CTRL-C` to stop the program. If we didn't do this, the ports would still be set when we forcefully exit the program.

line 23 sets up the "wait for the signal on port 23 to start falling towards 0"

lines 24-25 further on-screen instructions

line 27 cleans up the GPIO ports we've used during this program when `CTRL-C` is pressed
line 28 cleans up the GPIO ports we've used during this program when the program exits normally

Two ways to get the above code on your Pi

If you are in the command line on your Pi, type…

```
nano interrupt1.py
```
Then click “copy to clipboard” (above) and paste into the nano window.
```
CTRL+O
Enter
CTRL+X
```

Alternatively, you can download this directly to your Pi using…
```
wget http://raspi.tv/download/interrupt1.py.gz
gunzip interrupt1.py.gz
```

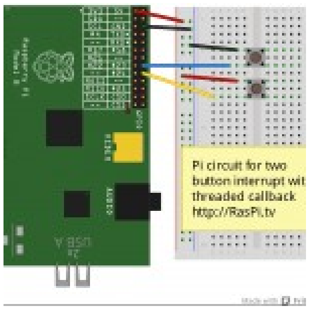
Then you can run it with…
```
sudo python
interrupt1.py
```

That’s cool, what next?

So that was a simple “wait for a button press” interrupt. There’s a lot more you can do with them, as I will show you in the next article, which will cover “threaded callback”, which allows us to use the spare capacity we’ve freed up by not polling continually.

How to use interrupts with Python on the Raspberry Pi and RPi.GPIO – part 2

raspi.tv/2013/how-to-use-interrupts-with-python-on-the-raspberry-pi-and-rpi-gpio-part-2


Pi circuit for two button interrupt wit threaded callback http://RasPi.tv

Interrupts are an efficient way for a program to be able to respond immediately to a specific event. In the previous article I explained the basics of using interrupts in RPi.GPIO and gave an example of a simple "wait for an event" interrupt program.

In this second article I will introduce "threaded callback" which opens up a lot of new possibilities.

Threaded callback – what the heck is that?

I know it sounds complicated. And it probably is complicated in the C code it's written in, but we're Pythonites and we don't have to go there. ;)

If you remember the previous example program was just a simple "wait for port 23 to be connected to GND when we press the button and then print a message and exit the program".

So, while it was waiting, the program wasn't doing anything else. The program only had one thread, which means only one thing was being done at once. Python is capable of running more than one thread at once. It's called multi-threading. It means that you can go through more than one piece of code simultaneously. This is where we can reap the benefit of interrupts because we can do something else while we wait for our "event" to happen. (Just like your "postman detector" allowed you to get on with something else instead of being distracted by waiting for the mail.)

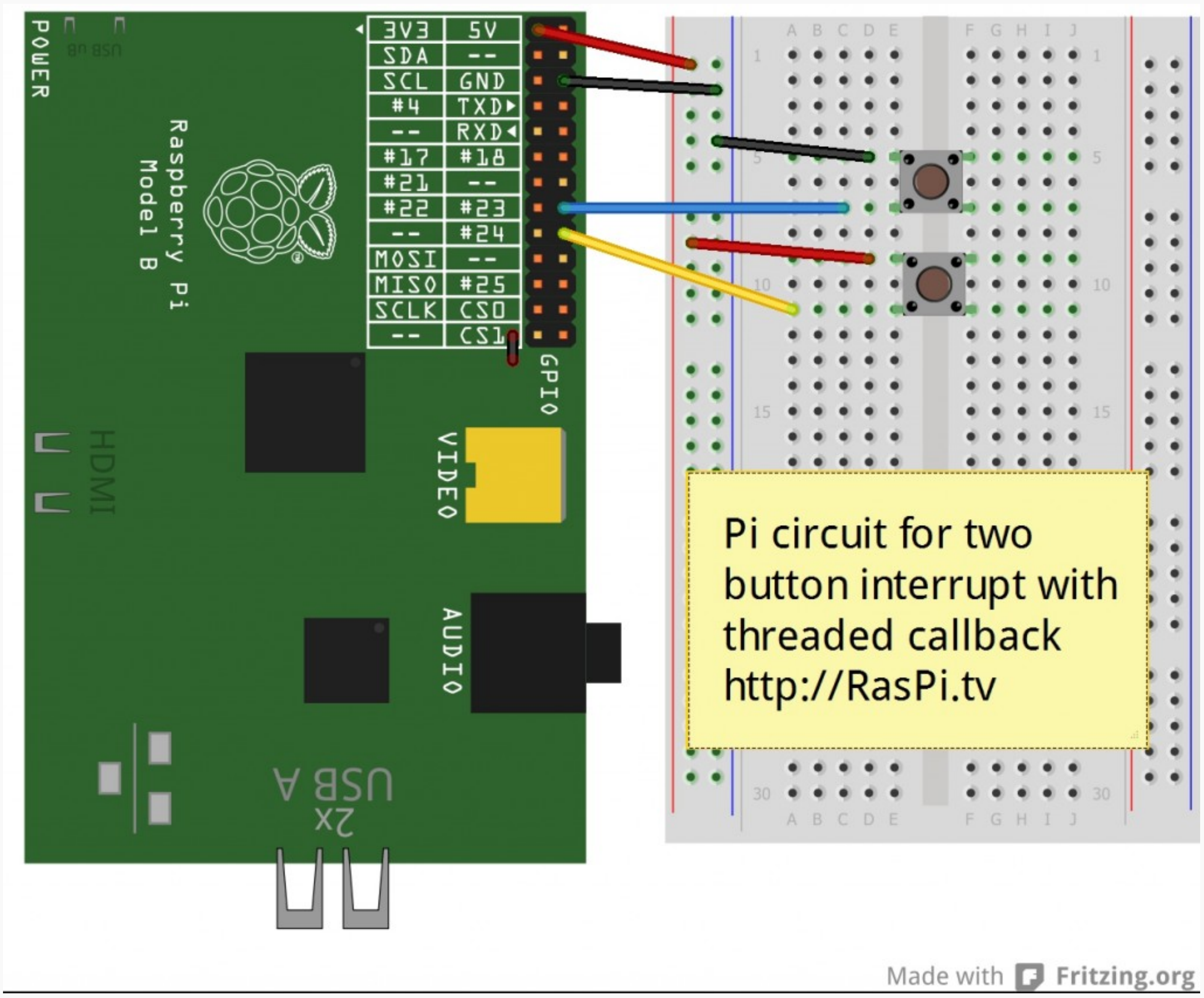
So that covers the threading part of threaded callback. What's a callback?
When an event is detected in the second thread, it communicates this back to the main thread (calls back). What we now have in RPi.GPIO is the ability to start a new thread for an interrupt and specify a set of instructions (function) that will run when the interrupt occurs in the second thread. This is a **threaded callback function.**

This is like your "postman detector" giving you a list of reminders of things you wanted to do when your delivery arrives AND doing them for you, so you can carry on with what you want to be doing.

So What are we going to do now?

We'll keep most of what we did before and add another button and an event detect threaded callback that runs when the new button is pressed, even though we are still waiting for the first button to be pressed.

But this time, the new button will connect GPIO port 24 to 3.3V (3V3) when pressed. This will allow us to demonstrate a rising edge detection. So we'll be setting up port 24 with the built in pulldown resistor enabled.



Circuit for second interrupt experiment

Later on we'll have to modify the code to cope with 'button bounce', but we won't say any more about that just yet.

Do you need to update RPi.GPIO?

If you didn't do it for the first example, you will quite likely need to update your RPi.GPIO package. You can check what version of RPi.GPIO you have in the command line with…

```
sudo python
import RPi.GPIO as
GPIO
GPIO.VERSION
```

This should show you what RPi.GPIO version you have. You need 0.5.2a or higher for this example.
You can exit the python environment with `CTRL+Z`

Install RPi.GPIO version 0.5.2 for simple interrupts

If you need to, you can install 0.5.2 or later with

```
sudo apt-get update
sudo apt-get dist-
upgrade
```

(This will update all your Raspbian packages and may take up to an hour)

Update July 2014

The best way to get the latest RPi.GPIO (currently 0.5.5) is to flash a new SD card with the latest NOOBS or Raspbian. This will give you a clean start with the latest version of RPi.GPIO.

And now onto the code

I've put most of the explanation in the code, so that if you use it, you will still have it.

```
#!/usr/bin/env python2.7
# script by Alex Eames http://RasPi.tv

import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

# GPIO 23 & 24 set up as inputs. One pulled up, the other down.
# 23 will go to GND when button pressed and 24 will go to 3V3 (3.3V)
# this enables us to demonstrate both rising and falling edge detection
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(24, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

# now we'll define the threaded callback function
# this will run in another thread when our event is detected
def my_callback(channel):
    print "Rising edge detected on port 24 - even though, in the main
thread,"
    print "we are still waiting for a falling edge - how cool?\n"

print "Make sure you have a button connected so that when pressed"
print "it will connect GPIO port 23 (pin 16) to GND (pin 6)\n"
print "You will also need a second button connected so that when pressed"
print "it will connect GPIO port 24 (pin 18) to 3V3 (pin 1)"
raw_input("Press Enter when ready\n>")

# The GPIO.add_event_detect() line below set things up so that
# when a rising edge is detected on port 24, regardless of whatever
# else is happening in the program, the function "my_callback" will be run
# It will happen even while the program is waiting for
# a falling edge on the other button.
GPIO.add_event_detect(24, GPIO.RISING, callback=my_callback)

try:
    print "Waiting for falling edge on port 23"
    GPIO.wait_for_edge(23, GPIO.FALLING)
    print "Falling edge detected. Here endeth the second lesson."

except KeyboardInterrupt:
    GPIO.cleanup()       # clean up GPIO on CTRL+C exit
GPIO.cleanup()           # clean up GPIO on normal exit
```

Two ways to get the above code on your Pi

If you are in the command line on your Pi, type…
`nano interrupt2.py`
Then click "copy to clipboard" (above) and paste into the nano window. Then
```
CTRL+O
Enter
CTRL+X
```

Alternatively, you can download this directly to your Pi using…

```
wget http://raspi.tv/download/interrupt2.py.gz
gunzip interrupt2.py.gz
```

Then you can run it with…

```
sudo python
interrupt2.py
```

What's supposed to happen?

When you run the code it gives you a message "Waiting for falling edge on port 23"
If you press button 1, it will terminate the program as before and give you a message
"Falling edge detected."

If, instead of button 1, you press button 2, you'll get a message
"Rising edge detected on port 24".

This will occur as many times as you press the button. The program is still waiting for the original falling edge on port 23, and it won't terminate until it gets it. Because your second button press is detected in another thread, it doesn't affect the main thread.

You may also notice, depending on how cleanly you press the second button, that sometimes you get more than one message for just one button press. This is called "switch bounce".

Bouncy, bouncy, bouncy

When you press a button switch, the springy contacts may flex and rapidly make and break contact one or more times. This may cause more than one edge detection to trigger, so you may get more than one message for one button press. There is, of course, a way round it, in software.

Why didn't this happen before?

I hear you ask. The answer is simple. Last time the program was simply waiting for a single button press. As soon as that button press was detected, it stopped waiting. So if the switch bounced, it was ignored. The program had already moved on. In our case, it had closed. But, when the event detection is running constantly in another thread, this is not the case and we actually need to slow things down a bit in what is called "software debouncing".

How to do software debouncing

In order to debounce, we need to be able to measure time intervals. To do that, we use the time module. Near the top of the program we need to add a line…

`import time` . I add it immediately after the RPi.GPIO import

```
import RPi.GPIO as
GPIO
import time
GPIO.setmode(GPIO.BCM)
```

Then, also quite near the top of the program we need to set an intial value for the variable time_stamp that we will use to measure time intervals

`time_stamp = time.time()` I put this after the GPIO.setup commands. This sets time_stamp equal to the time in seconds right now.

```
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(24, GPIO.IN,
pull_up_down=GPIO.PUD_DOWN)
time_stamp = time.time()
```

Now we have to change our threaded callback function from

```
def my_callback(channel):
    print "Rising edge detected on port 24 - even though, in the main
thread,"
    print "we are still waiting for a falling edge - how cool?\n"
```

to..

```
def my_callback(channel):
    global time_stamp       # put in to debounce
    time_now = time.time()
    if (time_now - time_stamp) >= 0.3:
        print "Rising edge detected on port 24 - even though, in the main
thread,"
        print "we are still waiting for a falling edge - how cool?\n"
    time_stamp = time_now
```

And now, even if you deliberately press the button twice, as long as you do it within 0.3 seconds it will only register one button press. You have debounced the switch, by forcing the program to ignore a button press if it occurs less than 0.3 seconds after the previous one.

If you are IDLE (Python joke) you can get the amended code here…

```
wget
http://raspi.tv/download/interrupt2a.py.gz
gunzip interrupt2a.py.gz
```

How does this work?

`time_now = time.time()` stores the current time in seconds in the variable time_now.
Now look at the last line of the function.
`time_stamp = time_now` This stores the time in seconds when the function was started in a global variable called `time_stamp` So next time this function is called, it will be able to check how much time has elapsed since the last time.

That's what is happening in the if statement.
`if (time_now - time_stamp) >= 0.3:` In English this means "If more than 0.3 seconds has elapsed since the last button press, execute the code in the indented block".

So if more than 0.3 seconds have elapsed it would print the message…
"Rising edge detected on port 24 – even though, in the main thread we are still waiting for a falling edge – how cool?"

If less than 0.3 seconds has elapsed, it will do nothing. Job done. Button switch is debounced.

Update – RPi.GPIO 0.5.2 onwards includes this debounce algorithm

Ben has included the above debounce algorithm in 0.5.2 onwards. This article was originally written for 0.5.1. So the above debounce code has been superseded by adding `bouncetime=xxx`, where xxx is a time in milliseconds. e.g.

```
GPIO.add_event_detect(channel, GPIO.RISING, callback=my_callback,
bouncetime=200)
```

I will update the next example to reflect this.

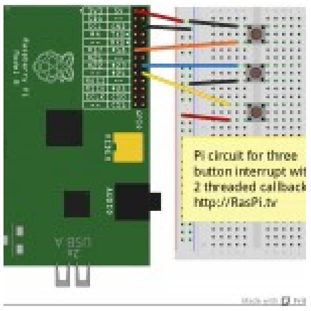
So, what's next?

1. We've learnt about simple "wait for" interrupts in the previous article
2. We've covered threaded callback in this article.
3. In the next article, we'll go over the situation where you want to do more than one threaded callback interrupt at once.

How to use interrupts with Python on the Raspberry Pi and RPi.GPIO – part 3

raspi.tv /2013/how-to-use-interrupts-with-python-on-the-raspberry-pi-and-rpi-gpio-part-3

Multiple threaded callback interrupts in Python



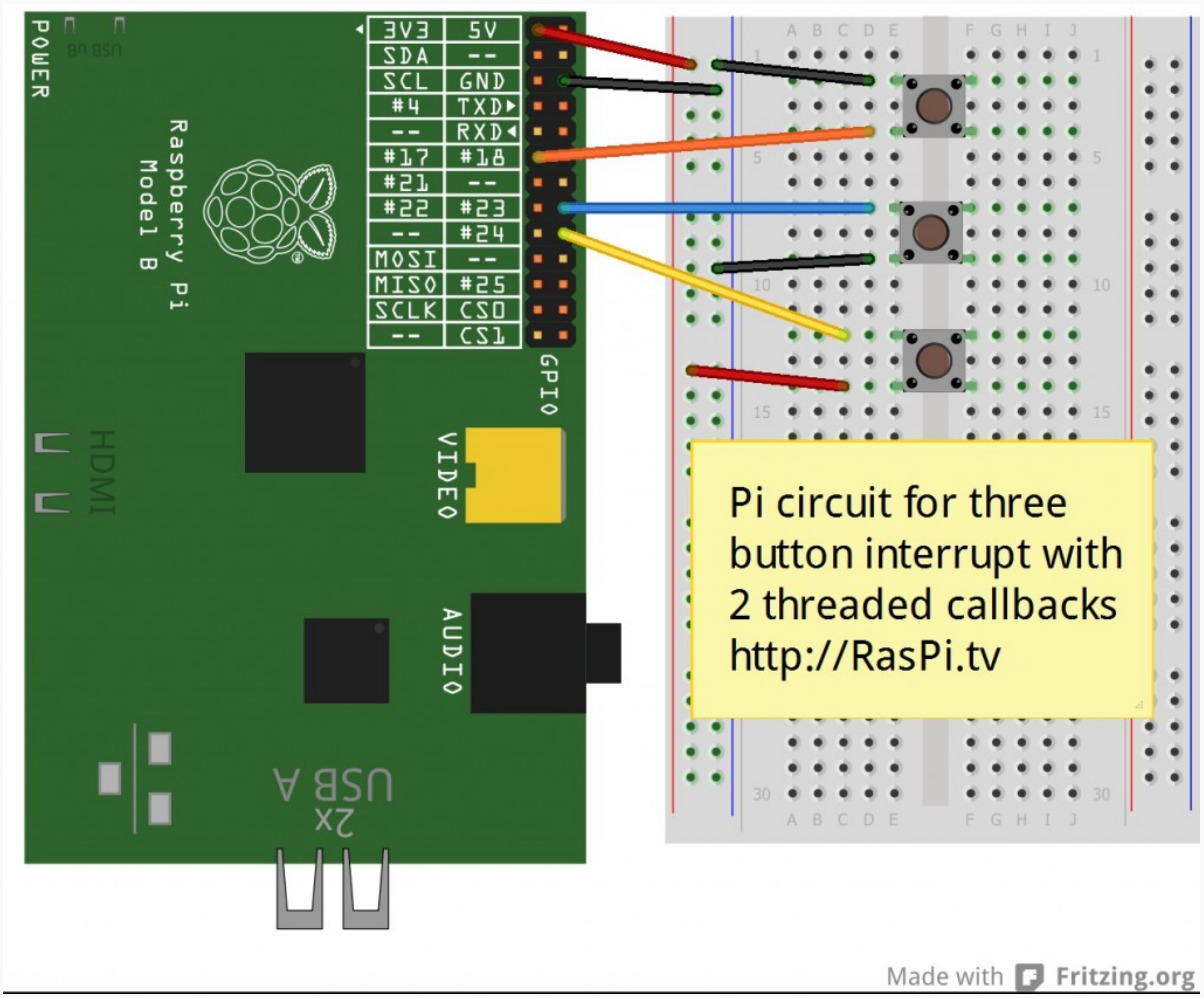
We've been learning about interrupts this week because of the brand new interrupt capabilities of RPi.GPIO. We covered a simple "wait for" interrupt in part 1, threaded callback interrupt and button debouncing in part 2 and today we're getting sophisticated with multiple threaded callbacks. "WooooooooOOOOOOOOOOOOOOOooooooooooo", I hear you say. ;)

Well actually, we're not doing much that's very different from last time, except, now there's more of it. We'll add another button and another threaded callback function the same as the first one (but on a different GPIO port). This is just to show that you can do multiple threaded callbacks in one program. After that, your imagination is the limit. (Well actually the number of GPIO ports is probably the limit.)

We're just building on what we did before and this is exactly how programs are made. You do a bit at a time, test it, fix it, make sure it does what it ought to do, then go on to the next bit.

Here's the Circuit

This circuit is a bit different from the previous one. The top two buttons connect port 17 and port 23 to GND when pressed. These are the two which trigger callbacks. The bottom button, connecting port 24 to 3V3 on button press is the "wait for" interrupt this time. So when you press button 3 it's "game over", but buttons 1 and 2 just report that they've been pressed until button 3 is eventually pressed.



Circuit for 2 threaded callbacks and one wait interrupt

We've used all the same building blocks we developed in parts 1 and 2, including button debouncing.

Do you need to update RPi.GPIO?

If you didn't do it for the first or second examples, you will quite likely need to update your RPi.GPIO package. You can check what version of RPi.GPIO you have in the command line with…

```
sudo python
import RPi.GPIO as
GPIO
GPIO.VERSION
```

This should show you what RPi.GPIO version you have. You need 0.5.1a or higher for this example.
You can exit the python environment with `CTRL+D`

Install RPi.GPIO version 0.5.2a for multiple threaded callback interrupts

If you need to, you can install 0.5.2a or later with

```
sudo apt-get update
sudo apt-get
upgrade
```

(This will update all your Raspbian packages and may take up to an hour)

Update July 2014

The best way to get the latest RPi.GPIO (currently 0.5.5) is to flash a new SD card with the latest NOOBS or Raspbian. This will give you a clean start with the latest version of RPi.GPIO.

And now onto the code

I've put most of the explanations in the code, so that if you use it, you will still have them.

```
#!/usr/bin/env python2.7
# script by Alex Eames http://RasPi.tv
# http://RasPi.tv/how-to-use-interrupts-with-python-on-the-raspberry-pi-and-rpi-
gpio-part-3
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)

# GPIO 23 & 17 set up as inputs, pulled up to avoid false detection.
# Both ports are wired to connect to GND on button press.
# So we'll be setting up falling edge detection for both
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(17, GPIO.IN, pull_up_down=GPIO.PUD_UP)

# GPIO 24 set up as an input, pulled down, connected to 3V3 on button press
GPIO.setup(24, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

# now we'll define two threaded callback functions
# these will run in another thread when our events are detected
def my_callback(channel):
    print "falling edge detected on 17"

def my_callback2(channel):
    print "falling edge detected on 23"

print "Make sure you have a button connected so that when pressed"
print "it will connect GPIO port 23 (pin 16) to GND (pin 6)\n"
print "You will also need a second button connected so that when pressed"
print "it will connect GPIO port 24 (pin 18) to 3V3 (pin 1)\n"
print "You will also need a third button connected so that when pressed"
print "it will connect GPIO port 17 (pin 11) to GND (pin 14)"
raw_input("Press Enter when ready\n>")

# when a falling edge is detected on port 17, regardless of whatever
# else is happening in the program, the function my_callback will be run
GPIO.add_event_detect(17, GPIO.FALLING, callback=my_callback, bouncetime=300)

# when a falling edge is detected on port 23, regardless of whatever
# else is happening in the program, the function my_callback2 will be run
# 'bouncetime=300' includes the bounce control written into interrupts2a.py
GPIO.add_event_detect(23, GPIO.FALLING, callback=my_callback2, bouncetime=300)

try:
    print "Waiting for rising edge on port 24"
    GPIO.wait_for_edge(24, GPIO.RISING)
    print "Rising edge detected on port 24. Here endeth the third lesson."

except KeyboardInterrupt:
    GPIO.cleanup()       # clean up GPIO on CTRL+C exit
GPIO.cleanup()           # clean up GPIO on normal exit
```

Bounce

`bouncetime=300` in lines 34 & 39 sets a time of 300 milliseconds during which time a second button press will be ignored. The code from example 2a has been incorporated into RPi.GPIO.

You can download this directly to your Pi using…

```
wget http://raspi.tv/download/interrupt3.py.gz
gunzip interrupt3.py.gz
```

Then you can run it with…

```
sudo python
interrupt3.py
```

Can I switch off an event detection?

There's just one more thing. If you want to stop an event detection on a particular port, you can use the following command…

```
GPIO.remove_event_detect(port_number)
```

You now know all you need to know about how to create and manage interrupts in RPi.GPIO in Python on the Raspberry Pi. The official documentation is here if you want to check it out.

Have fun with interrupts. I hope this mini-series has been useful. Let us know, in the comments below, what you're planning to use it for and then come back and tell us how it went. :)

For extra credit, can you tell me why I chose those specific GPIO ports for these experiments?